Experimental Results for Class-Based Queueing

Sally Floyd* and Michael Francis Speer

October 16, 1997

1 Introduction

This paper describes the experience of implementing CBQ's top-level link sharing code as described previously in [FJ95]. This implementation incorporates work from previous implementations of Class-Based Queueing from LBNL [Jac95] and UCL [WGC+95]. This implementation extends the work of previous implementations by incorporating both Top-Level link-sharing and Weighted Round Robin within priority levels of the link-sharing structure.

As discussed in [FJ95] and [Flo97], the use of Top-Level instead of Ancestor-Only link-sharing allows a class to receive its allocated bandwidth more accurately from a CBQ implementation.

Similarly, as discussed in [FJ95] and [Flo97], weighted round robin (WRR) has two advantages over packet-by-packet round robin (PRR) scheduling within a priority level. First, WRR gives better worst-case delay behavior than PRR scheduling for higher-priority classes. Second, WRR scheduling allows excess bandwidth to be distributed among classes in a priority level according to the bandwidth allocations of those classes.

2 Description of CBQ Implementation

2.1 General CBQ description

Before discussing this CBQ implementation in detail, it is important to note that this CBQ implementation is built utilizing many components. At the high level CBQ is not just a packet scheduler; it is a link-sharing resource manager. In principle, CBQ's link-sharing could be implemented in conjunction with a number of different packet scheduling algorithms within a priority level, such as Deficit Round Robin, Weighted Fair Queueing, or Fair Queueing. This implementation utilizes an implementation of Weighted Round Robin (WRR) and/or Packet-by-Packet Round Robin (PRR) scheduling. Compared to other general scheduling algorithms, these two schedulers seem to be the least expensive in computational complexity. We discuss the details of these scheduling algorithms later in the paper.

*This work was supported by the Director, Office of Energy Research, Scientific Computing Staff, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

2.2 Principles Applied in CBQ Implementation

This CBQ implementation follows many principles previously outlined in [FJ95] to allow for the maximum flexibility.

First, this CBQ implementation continues to maintain a separation of low-level mechanisms and high-level policy. The CBQ kernel code provides a rich interface to implement variety of high-level policies, including, if so desired, RSVP and Integrated Services.

Second, the CBQ implementation preserves the link-sharing model presented in [FJ95]. Link-sharing resources are associated with CBQ traffic classes, where each CBQ traffic class has a bandwidth allocation and a priority. It is left to the high-level policy daemon to make decisions about *how* to allocation bandwidth and priorities to the various classes. The CBQ implementation is able to handle Quality-of-Service (QOS) and link-sharing constraints simultaneously.

Finally, the CBQ implementation avoids the need for extensive per-conversation parameterization. Hence, this CBQ implementation is able to separate IP conversations (flows) into classes with a minimum of class and filter parameterization.

2.3 CBQ Implementation

Major components of this implementation for CBQ's top-level link sharing include a packet classifier, link-sharing framework, packet scheduler, estimator, and management interface. The packet classifier maps arriving packets into traffic classes. The link-sharing framework is needed to maintain link-sharing constraints for an interface (e.g. an output port) with a hierarchical link-sharing structure. The packet scheduler schedules traffic classes according to their bandwidth and priority considerations, with help from the estimator. The management interface allows for the creation and deletion of traffic classes; the creation and deletion of packet classifier filters to map IP flows to the appropriate traffic classes; and a simple statistical interface for inspection of CBQ's current state.

In this CBQ implementation, the link-sharing framework is implemented using Top-Level Link-Sharing, described in [FJ95]. Previous implementations implemented Ancestor-Only link-sharing.

In this CBQ implementation, the packet scheduler or selector is implemented with either a packet-by-packet round robin (PRR) or weighted round robin (WRR) scheduler. The scheduler uses priorities, first scheduling packets from the highest priority level. Round-robin scheduling is used to arbitrate between traffic classes within the same priority level. The weighted round robin scheduler differs from the packet-by-packet scheduler in that it uses weights proportional to a traffic class's bandwidth allocation. The weight determines the number of bytes that a traffic class is allowed to send during a round of the scheduler. If a packet to be transmitted by a WRR traffic class is larger than the traffic class's weight and the class is underlimit (via link-sharing constraints), then the packet is sent, allowing the traffic class to borrow ahead from its weighted allotment for future rounds of the round-robin. The implementation of the WRR scheduler largely follows that of the CBQ code in the "ns" simulator [MF95]. In the implementation of the CBQ code the scheduler components are implemented as the functions *_rmc_prr_dequeue_next* for PRR and *_rmc_wrr_dequeue_next* for WRR, both in the file rm_ class.c [FS97].

When a traffic class is overlimit and unable to borrow from parent classes, the scheduler activates the overlimit action handler for that class. There are many policies that could implemented for an overlimit class, including simply dropping arriving packets for such a class. This CBQ implementation rate-limits overlimit classes to their allocated bandwidth. The rate-limiter computes the next time that an overlimit class is allowed to send traffic. The class will not be allowed to send another packet until this future time has arrived. This rate-limiter action is implemented as *rmc_delay_action* in the file rm_class.c of the CBQ implementation [FS97].

The estimator estimates the bandwidth used by each traffic class over the appropriate time interval (or, more precisely, simply estimates whether each class is over or under its allocated bandwidth). As discussed later, the time constant for the estimator determines the interval over which the router attempts to enforce the link-sharing bandwidth constraint. Hence the parameterization of this time constraint is key to enforcing link-sharing bandwidth allocations. This implementation employs a exponential weighted moving average (EWMA) to estimate the bandwidth used by each class. In this CBQ implementation, the estimator is implemented as the function *rmc_update_util* in the file rm_ class.c of the CBQ implementation [FS97].

The network management interface for this CBQ implementation allows for RSVP [BZ97] and other resource management mechanisms to configure the output link in the manner appropriate for those mechanisms. The network management interface allows for the creation and destruction of CBQ traffic classes, the appropriate filters to map the IP flows to traffic classes via the packet classifier, and a rather crude statistical interface for monitoring CBQ's internal state. All code in the management interface can be found in the file cbq.c of this CBQ implementation [FS97].

3 CBQ Parameters

The CBQ parameters for each class are set at class creation time. Using the experimental policy daemon cbqd, classes are created and parameterized as specified in a configuration file. Each class definition supplies the *priority*, the allocated *bandwidth* for the class (expressed in terms of link bandwidth percentage), *average packetsize*, *maxburst*, *minburst* and *maxdelay*. *Average packetsize* is used in calculating *maxidle* and *offtime*, as shown below. *Maxburst* is the maximum burst size for the class (that is, the maximum number of back-to-back packets sent by a previously-idle class). *Minburst* is the burst size for an overlimit class that is being regulated to its allocated bandwidth. *Maxdelay* is the target maximum delay (in milliseconds) that *average packetsize* packets will have to wait to be scheduled. *Maxdelay* is used to determine the *maxq* (maximum queue length in number of packets) parameter for the CBQ traffic class. Using these class parameters, other class parameters such as maxidle are derived to drive the CBQ scheduling apparatus.

For each class parameter not supplied in the class definition, default values are supplied. Some of these default values are as follows: *maxburst* defaults to 20 packets; *minburst* defaults to 2 packets; *average packetsize* defaults to 1000 bytes; and *maxdelay* defaults to 100 milliseconds.

In the CBQ policy daemon associated with the distributed code [FS97], the function *cbq_create_class* in the file cbqif.c utilizes the various inputs to compute the parameters discussed below.

3.1 CBQ Parameter Definitions

In CBQ, each class has variables *idle* and *avgidle*, and a parameter *maxidle* used in computing the limit status for the class. This section discusses setting the *maxidle* parameter. At one time a *minidle* parameter was used in the ns simulator, but that parameter has been removed, and there is now no lower bound on *avgidle*.

Definition: idle. The variable *idle* is the difference between the desired time and the measured actual time between the most recent packet transmissions for the last two packets sent from this class. When the connection is sending perfectly at its allotted rate p, then *idle* is zero. When the connection is sending more that its allocated bandwidth, then *idle* is negative.

Definition: avgidle. The variable *avgidle* is the average of *idle*, and is computed using a exponential weighted moving average (EWMA). When *avgidle* is zero or lower, then the class is overlimit (the class has been exceeding its allocated bandwidth in a recent short time interval).

Definition: maxidle. The parameter *maxidle* gives an upper bound for *avgidle*. Thus *maxidle* limits the `credit' given to a class that has recently been under its allocation.

Definition: offtime. The parameter *offtime* gives the time interval that a overlimit class must wait before sending another packet. This parameter is determined in part by the steady-state burst size *minburst* for a class when the class is running over its limit. In the ns simulator [MF95], this steady-state burst size is controlled by the *extradelay* parameter. A steady-state burst size of one packet can be achieved in the ns simulator by setting setting *extradelay* to 0. In the CBQ implementation a small steady-state burst size is achieved by setting *minburst* to 1.

3.2 Setting Maxidle

Maxidle controls the burstiness allowed to a class. As Appendix **??** shows, to permit a maximum burst of $maxburst$ back-to-back packets, *maxidle* is set as follows:

$$maxidle \leftarrow t(1/p-1)\frac{1-g^{maxburst}}{g^{maxburst}},$$

for t the interpacket time for `average' sized packets sent back-to-back, p the fraction of the link bandwidth allocated to the class, and weight g, for

$$g = 1 - 2^{-RM\_FILTER\_GAIN}$$

$$= 1 - 1/RM\_POWER.$$

In addition, the following constraint should be observed:

$$maxidle \geq t(1-g).$$

Appendix **??** shows that the calculation of *avgidle* in the code in fact corresponds to the equations in [FJ95]. Appendix **??** justifies the equations used for setting the variable *undertime* in the procedure *rmc_update_class_util*.

3.3 Setting Offtime

For leaf classes, *offtime* controls the steady-state burst size for a regulated class. In cbqd, for a regulated class with a burst size of 1, *offtime* in its unscaled value is set as follows:

$$offtime \leftarrow t(1/p-1).$$

This is the target waiting time to maintain the allocated bandwidth with a steady-state burst size of only one packet.

For a steady-state burst size of $minburst + 1$ packets for $minburst \geq 1$, cbqd further modifies *offtime* as follows:

$$offtime \leftarrow offtime * \left(1 + \frac{1}{(1-g)}\frac{(1-g^{minburst})}{g^{minburst}}\right).$$

4 Description of experimental testbed

In the development and the testing of this CBQ implementation, one testbed was used to test and refine the CBQ implementation and collect the results from various experiments. The testbed as seen in Figure 1.

This testbed employs 6 Sun SPARCstation 20 and Sun SPARCstation 5 workstations. The router in figure 1 is a Sun

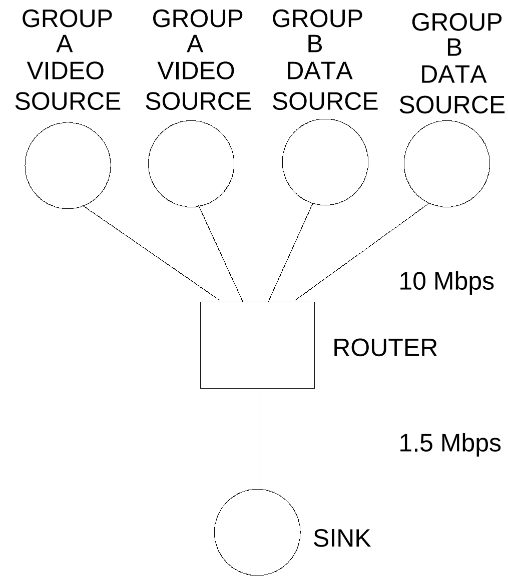

Figure 1: Network Setup for Link Sharing Experiments

SPARCstation 20 with two 125 MHz HyperSPARC CPUs with 256 KBytes of external cache. The router running Solaris 2.5.1 has been updated with TCP/IP kernel modules that will accommodate RSVP operation, routing functionality, IGMPv2, and DVMRP multicast routing. Within this testbed, *cbqd* was employed to configure the link between the router and the sink to test the CBQ implementation in various link sharing experiments. Each of the sources in the testbed where connected to the router via switched ethernet on individual networks.

For testing RSVP operation with CBQ, the testbed in figure 1 was upgraded. The link between the router and the sink was upgraded to 10 MBit/second switched ethernet. All the links between the router and the sources remained the same.

In performing CBQ and/or RSVP experiments, a number of parameters were captured and examined to gain insight on the performance of the CBQ machinery. These parameters included *packet delay*, *throughput*, *packet drops*, and *avgidle* within a class. To capture these parameters, a number of tools where employed including *tcpdump* and *adb*.

References

[BZ97] R. Braden and L. Zhang. Resource reservation protocol version 1 functional specification. (Internet draft, work in progress), June 1997. URL ftp://ds.intenic.net/internet-drafts/draft-ietf-rsvp-spec-16.txt.

[FJ95] S. Floyd and V. Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4), 1995.

[Flo97] S. Floyd. Ns simulator tests for class-based queueing. *Unpublished draft*, Apr. 1997. URL ftp://ftp.ee.lbl.gov/papers/cbqsims.ps.Z.

[FS97] S. Floyd and M. Speer. Lbnl's cbq code v2.0. Technical report, May 1997. URL ftp://ftp.ee.lbl.gov/cbq2.0.tar.Z.

[Jac95] V. Jacobson. Lbnl's cbq code v1.1. Technical report, August 1995. URL ftp://ftp.ee.lbl.gov/cbq.tar.Z.

[MF95] S. McCanne and S. Floyd. Ns (network simulator), 1995. URLs http://www-nrg.ee.lbl.gov/ns, http://www-mash.cs.berkeley.edu/ns/.

[WGC+95] I. Wakeman, A. Ghosh, J. Crowcroft, V. Jacobson, and S. Flo yd. Implementing real time packet forwarding policies using streams. Technical report, January 1995. URL ftp://cs.ucl.ac.uk/darpa/usenix-cbq.ps.Z.